SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 117–144 (1996)

Research

Reverse Engineering Distributed Algorithms

K. SERE

University of Kuopio, Department of Computer Science and Applied Mathematics, P.O. Box 1627, SF-70211 Kuopio, Finland

M. WALDÉN

Aabo Akademi University, Department of Computer Science, SF-20520 Turku, Finland

SUMMARY

Distributed systems are difficult for a human being to comprehend. Informal reasoning about the many parallel and decentralized activities in these systems is not trustworthy. Therefore formal tools for construction and maintenance of distributed systems are needed. We introduce a *formal approach to reverse engineering distributed systems* that is based on a technique we call *coarsement*. The idea is that an implementation is stepwise turned into a high level specification through a number of intermediate coarsement steps that preserve the basic functionality of the implementation. The method gives structure to a distributed algorithm that can now be seen as consisting of a number of layers interacting with each other. Each coarsement step produces one such layer. Furthermore, after the coarsement steps the algorithm is easier to understand and to reason about than the original one due to this layering. We show the practical feasibility of the coarsement approach to reverse engineering by analysing a non-trivial distributed algorithm that maintains the routeing information for message passing among a set of processing nodes in a distributed network.

KEY WORDS: reverse engineering; distributed systems; tools and techniques; specifying and verifying and reasoning about programs

1. INTRODUCTION

The process of analysing a system in order to identify its components and their inter-relationships is called *reverse engineering*. One result of the analysis could be a high level specification of the system which makes the system more comprehensible. Since understanding software is time consuming, it is also costly. By reducing the time required to understand programs, reverse engineering may greatly reduce the overall cost of software (Chikofsky and Cross, 1990).

Recently, formal approaches to reverse engineering have received considerable attention as a means of creating correct high level specifications (Breuer and Lano, 1991; Ward, 1993). It is, however, argued that these methods are too strict and not applicable in real life situations (Brown, 1993).

In this paper we show that formal approaches to reverse engineering are indeed

CCC 1040–550X/96/020117–28

Received December 1994
Revised November 1995

applicable when constructing high quality *distributed systems*. Distributed systems are difficult for a human being to comprehend. Informal reasoning about the many parallel and decentralized activities in these systems is not trustworthy. Therefore formal tools are needed. Moreover, for distributed systems the *correctness* of the algorithms is perhaps the most important characteristic to preserve and/or establish. Without formal tools convincing proofs about the correctness of distributed programs are not possible to produce. However, many distributed algorithms exist and are in use today. For many of them formal proofs have not been presented. By using the techniques presented in this paper programmers and systems analysts developing distributed programs can more easily understand existing undocumented code and have a tool to create abstract specifications and proofs for these programs.

We introduce a formal approach to reverse engineering distributed systems. Our approach is based on a technique we call *coarsement*. The idea is that an implementation is stepwise turned into a high level specification, a more coarse-grained system, through a number of intermediate coarsement steps that preserve the basic functionality of the implementation. Hence, it is our intention to mimic the usual stepwise refinement approach to program construction, but in an opposite direction.

The stepwise refinement approach to program construction has been previously formalized as the *refinement calculus* (Back, 1978; Morgan, 1988; Morris, 1987). Originally this calculus was developed for the derivation of provably correct sequential programs. The calculus has been extended since then and now a refinement calculus for parallel and distributed systems exists (Back and Sere, 1993; Sere, 1990). We will formalize the coarsement method within this extended calculus. Hence, we show that a formal calculus designed for the derivation of correct programs can be used as a basis for formal program maintenance.

We develop a method and a program transformation rule that allow us to formally reason about the coarsement steps. At each coarsement step some mechanism of the algorithm is identified and removed from the rest of the code using the transformation rule. The method therefore gives more structure to the distributed algorithm that can now be seen as consisting of a number of layers interacting with each other. Each coarsement step produces one such layer. Furthermore, after the coarsement steps the algorithm is easier to understand and to reason about than the original one due to this layering.

The result of carrying out the coarsement steps is a high level specification of the algorithm. The correctness of the specification with respect to the original algorithm can be established, because each coarsement step can be formally verified within the refinement calculus using the developed transformation rule. Furthermore, by giving a proof of correctness for the high level specification we obtain a proof of correctness of the entire algorithm.

We show the practical feasibility of the coarsement approach to reverse engineering by analysing a non-trivial distributed algorithm due to Tajibnapis (1977). This important algorithm maintains the routeing information for message passing among a set of processing nodes in a distributed network, a central task in a distributed system. Using the coarsement method we show that this algorithm can be restructured into a rather small kernel containing the basic functionality accompanied with a number of layers where each layer optimizes and distributes the basic computation.

Overview of the paper

In Section 2 we describe our approach to reverse engineering and give an initial example of the coarsement method. Section 3 deals with the formalization of coarsement within the refinement calculus for *action systems*, which is the formalism we use to model parallel and distributed algorithms (Back and Kurki-Suonio, 1983). The algorithm of Tajibnapis is discussed in Section 4. The coarsement steps that we carried out are described in Section 5. Finally in Section 6 we give some comments on the coarsement method.

2. AN APPROACH TO REVERSE ENGINEERING

In this section we informally describe our approach to reverse engineering distributed systems and give a first example of the coarsement method, which is the most central part of the approach.

In coarsement modifications we, for example, identify a distributed data structure and change it to a variable that is shared between the processing elements. We can also abstract away some information gathering mechanism from the system and thereby introduce direct access to some shared variable. The result is that the grade of atomicity often diminishes due to the coarsement steps and the algorithm becomes more abstract, i.e., we develop a more coarse-grained system.

The changes to an algorithm are done in such a way that the underlying computation is not disturbed while some functionality is abstracted away from the code. The motivation behind such modifications is that shared variables and centralized control are often easier to reason about than distributed data structures and decentralized control.

2.1. The method

The method we propose basically consists of identifying the data structures of an algorithm and thereafter making the algorithm more abstract by removing some data structures while expressing their contribution to the overall computation with the remaining data structures.

When reverse engineering a distributed algorithm, we proceed as follows:

(1) We identify the central data structures of the algorithm and find an invariant and/or a post-condition on these data structures that describe *what is to be computed* by the algorithm.
(2) We identify the data structures relevant to the central ones which describe *how the central computation is performed.*
(3) We repeat the following steps until a high level specification of the algorithm has been obtained. At each step we concentrate on one data structure identified in step 2.
    (i) We determine the data structure to be abstracted away at this step and express its invariant as a relation between the removed and the remaining variables.
    (ii) We remove the data structure (determined in step (i)) expressing its effect on the computation in terms of the remaining variables using the invariant.
    (iii) We show that the behaviour of the program in terms of the remaining variables is still preserved with respect to the data structures that have been abstracted away.

Steps 1, 2 and 3(i) are performed by inspecting the informal specification and the code of the algorithm. These steps require creative thinking, because we need to develop invariants that define the relevant data structures. The invariant we find in step 1 states a condition on the central data structures giving some general properties for these structures. In step 3(i) the invariant for the data structure to be abstracted away expresses the relationship between the removed variable and the remaining variables in detail. Steps 3(ii) and (iii) can be performed more mechanically.

We call step 3 the *coarsement* step. During the coarsement steps we structure the program into some basic computation accompanied by a collection of mechanisms that perform the task of this basic computation in an efficient and distributed manner. The basic computation is expressed in terms of the variables identified in step 1. At each coarsement step one mechanism, expressed in terms of the data structures in step 2, is identified.

When formalizing the method, we rely on the refinement calculus. This gives us a possibility to directly work with the program text on a formal basis, which is convenient especially at step 3. However, the refinement calculus also guides us in steps 1 and 2 as it helps in finding the invariant and in identifying the data structures.

In addition to analysing and structuring an algorithm, we can also use the method as a way of *verifying* distributed algorithms, because at each step we are able to formally verify the correctness of the modifications. Furthermore, using the standard invariant based verification methods (Dijkstra, 1976), the correctness of the high level specification can be established. The verification aspect is studied more thoroughly in (Sere and Waldén, 1994).

2.2. Example: minimum-hop distances

As an example of reverse engineering distributed algorithms using the coarsement approach we consider an algorithm that computes so called minimum-hop distances in a network of processing nodes. The minimum-hop distance between two nodes x and y in a network denotes the length of the path from x to y which goes via a minimum number of intermediate nodes.

Let (V, E) be a connected graph with V a finite set of nodes and E a finite set of edges on V. Let the nodes denote processes and the edges denote communication links between the processes. Communication can only take place between neighbouring nodes, i.e., nodes directly connected by a bidirectional edge. Each process is assumed to know the identities of its direct neighbours in the network.

In Figure 1 we give an example of a network (a graph), where node i has nodes j

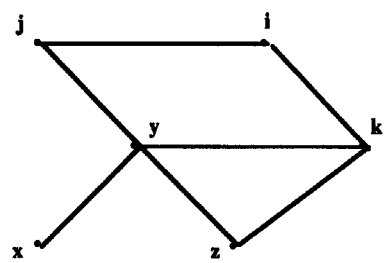

Figure 1. Example of a network computing minimum-hop distances

and k as neighbours. Let δ denote the correct minimum-hop distances between nodes. In Figure 1 the minimum-hop distance $\delta.j.i$ from node j to node i is 1 and $\delta.j.x$ from node j to node x is 2. The minimum-hop distance between two nodes need not follow a unique path. For example, the minimum-hop distance $\delta.i.x$ between node i and node x is 3. There are, however, two paths that satisfy this, one path via the nodes j and y and the other via the nodes k and y.

The minimum-hop distance $\delta.i.x$ between every pair of nodes i and x is formally defined as follows:

$$(\forall i, j, x \in V : (i,j) \in E,\ x \neq i :$$
$$(0 \leqslant \delta.i.x \leqslant N \wedge \delta.i.i = 0 \wedge \delta.i.x = \min\{N, (\delta.j.x + 1 \mid (i,j) \in E)\})$$

Here N denotes the number of nodes. The longest possible distance between two nodes is $N - 1$, while N denotes the length of infinite distances. A distance between two nodes is infinite, if no path exists between them. Thus, the minimum-hop distance between two nodes is between 0 and N, where the minimum-hop distance from a node to itself is 0. Furthermore, the distance $\delta.i.x$ between two arbitrary nodes i and x is one longer than the shortest distance $\delta.j.x$ of a neighbouring node j to x.

The program $\mathcal{S}$ in Figure 2 calculates in a distributed fashion the minimum-hop distances between every pair of nodes in any graph (V, E), where the number of nodes N is greater than 1. Each node i holds the computed distance in variable $D.i.x$ for every node x. Let us now analyse this algorithm.

Note on notation

We use the multiple assignment statement as a convenient notation for working with lists (as links will be represented with lists). If x is a variable of type *value* and Q is a variable of type *value list*, then $x, Q := Q$ will assign the first element of Q to x and

```
S =  |[ var Q.i.j value list  for i,j ∈ V;
        D.i.i := 0  for i ∈ V; D.i.j := N  for i,j ∈ V(i ≠ j);
        Q.i.j :=<>   for i,j ∈ V(i ≠ j);                              (1)
        do
[A.i.j]  | D.i.j > 1 →
             D.i.j := 1;
             Q.i.a := Q.i.a, < j,1 >   for (i,a) ∈ E                  (2)
        for (i,j) ∈ E
[B.i.j]  | Q.j.i ≠ <> ∧ hd(Q.j.i) = < x,l > ∧                         (3)
                x ≠ i ∧ x ≠ j ∧ D.i.x > l + 1 →
             < x,l >, Q.j.i := Q.j.i; D.i.x := l + 1;                 (4)
             Q.i.a := Q.i.a, < x, D.i.x >   for (i,a) ∈ E             (5)
        for (i,j) ∈ E
[C.i.j]  | Q.j.i ≠ <> ∧ hd(Q.j.i) = < x,l > ∧                         (6)
              (x = i  ∨  (x ≠ i ∧ x ≠ j ∧ ¬(D.i.x > l + 1))) →
             < x,l >, Q.j.i := Q.j.i                                  (7)
        for (i,j) ∈ E
        od
    ]|: D.i.j ∈ int  for i,j ∈ V
```

Figure 2. An algorithm calculating minimum-hop distances

remove it from Q. Similarly, $Q := Q.x$ will add x as last element to Q. Furthermore, we use the **for**-clause to state that the previous declaration or statement is replicated once for each value of the index variable.

Reverse engineering

In order to perform a coarsement step we proceed according to our method as follows:

(1) First we identify the central data structures. In the algorithm $\mathcal{S}$ the distance array D is the central data structure, for which the condition R below should hold at termination:

$$R : (\forall i, x \in V : D.i.x = \delta.i.x)$$

This predicate states that all the nodes i have correct minimum distances $\delta.i.x$ to every other node x.

By inspecting the code we find that the following invariant holds during the computation when the number of nodes in the network is greater than one, $N > 1$:

$$I : (\forall i, x \in V : 0 \leq D.i.x \leq N)$$

because initially each node sets the distance to itself to 0 and the distance to all other nodes to N. Furthermore, the distances between nodes are only decreased during the computation due to the following. The distance to a neighbouring node is set to 1 in the command $A.i.j$. (We have given labels to the guarded commands for ease of reference.) This distance is distributed to all the neighbours using the variable Q:

$$Q.i.a := Q.i.a, <j,1> \textbf{ for } (i,a) \in E$$

A node i only changes its distances $D.i.x$ upon receiving a new shorter distance to destination x from a neighbouring node than it already has. This takes place in command $B.i.j$, where we require that $D.i.x > l + 1$ for some received distance l. After decreasing $D.i.x$ it distributes this distance to its neighbours again via the variable Q:

$$Q.i.a := Q.i.a, <x, D.i.x> \textbf{ for } (i,a) \in E$$

The command $C.i.j$ does not affect the array D.

(2) The data structures relevant to the central data structure D are also found by inspecting the code. The data structure Q is such a structure. The variable $Q.i.j$ denotes the communication link from node i to node j. The messages sent over the links are of the form $<x,l>$ where $x \in V$ is a node and l is an integer, some computed distance from the sender of the message to the node x.

In the algorithm $\mathcal{S}$ the sending of a message from node i to node j is denoted by assigning a message to $Q.i.j$. The fact that node i receives a message from node j is stated by removal of a message from $Q.j.i$.

Let us take a closer look at the algorithm in Figure 2. After initializing $D.i.i$ to 0 and Q to the empty list, the minimum-hop calculation is started by a node i setting its distance to a neighbour j to 1 and sending this distance to all neighbours. This takes place in the guarded command $A.i.j$ as described above in step 1.

The guarded commands $B.i.j$ and $C.i.j$ are executed whenever there is a message from node j to node i available along link $Q.j.i$. Let us consider these commands more carefully. If $x \neq i$ in $B.i.j$, node i computes a new distance to node x in case the old distance stored in $D.i.x$ is greater than the received distance l plus 1, i.e. $D.i.x > l + 1$. The updated distance is stored in $D.i.x$ and subsequently sent to every neighbour. If the received distance l in $C.i.j$ is greater than the current distance $D.i.x$ and $x \neq i$ or if $x = i$, the message is ignored and removed from the link.

A coarsement step

Let us now make a coarsement step on $\mathcal{S}$, continuing with step 3 of our reverse engineering method.

(i) We abstract away the links Q that exist between every pair of nodes in the network. (All computation that involves variable Q is denoted with `this font` in Figure 2. Furthermore, every line where this modification will take place carries a label that has been indicated in the figure.) This coarsement step is motivated by the fact that the links are not needed for the computation of the distances, if every node is able to compare its distance information to other nodes directly with its neighbours.

The invariant P formally expresses the removed mechanism Q with respect to the remaining variables as follows

$$P\colon P_1 \wedge P_2 \wedge P_3$$

where P_1, P_2 and P_3 are given below:

$$\begin{aligned}
P_1\colon\ & (\forall i, j, x \in V, l \in int\colon (i,j) \in E, x \neq j\colon \\
& \quad <x,l> \in Q.j.i \Rightarrow (0 \leq l \leq N) \wedge (D.j.x \leq l)) \\
P_2\colon\ & (\forall i, j, x \in V, l \in int\colon (i,j) \in E, x \neq i, x \neq j\colon \\
& \quad (D.j.x = l \wedge l < N) \Rightarrow (\exists k \in int\colon Q.j.i.k = <x,l>)) \\
P_3\colon\ & (\forall i, j, x \in V\colon (i,j) \in E, x \neq i, x \neq j\colon \\
& \quad (\exists l \in int\colon <x,l> \in Q.j.i \wedge D.i.x > l + 1) \equiv D.i.x > D.j.x + 1)
\end{aligned}$$

The invariant P_1 states that the distances l that are sent on the links should be between 0 and N. Furthermore, it states that the distances can only decrease, never increase. The current $D.j.x$ in node j is thus less or equal to all the distances l sent from node j. According to the invariant P_2 a node always sends a computed distance to all of its neighbours. Moreover, the invariant P_3 states that the computed distance $D.i.x$ to a node x from node i is greater than the distance to x plus 1 via a neighbouring node j, $D.j.x + 1$, if and only if there is a message on the link from node j to node i with a new shorter distance to x.

In the invariant P_2 we consider all the messages that have been sent over a link during a computation. Here k is a pointer to a message on the link. This variable is not found in the algorithms $\mathcal{S}$ and $\mathcal{S}'$. It is only needed for the proofs.

(ii) This coarsement gives us the algorithm $\mathcal{S}'$ in Figure 3, where the commands $B.i.j$ and $A.i.j$ in $\mathcal{S}$ are changed to $B'.i.j$ and $A'.i.j$ in $\mathcal{S}'$ respectively relying on the invariants. The command $C.i.j$ is abstracted away. In the initialization and in the action $A'.i.j$ only the references to Q (changes (1) and (2)) are removed relying on the invariants P_1 and P_2, where P_1 states that the links are well-behaved and contain the right type of messages. According to P_2 the distances of a node are available on the links. Also in $B'.i.j$ the links are removed relying on P_1 and P_2 as above (change (5)). Furthermore, the remaining code is slightly modified due to invariant P_3: the distance l from node j to node x that is used in $B.i.j$ is replaced with a direct reference to the distance $D.j.x$ of neighbour j to node x in $B'.i.j$ (changes (3) and (4)). The action $C.i.j$ only contains reading references to Q and can thus be removed (changes (6) and (7)).

(iii) The final step in a coarsement modification is the verification of the correctness of the transformations. For this we need a more formal way to apply our method. Therefore, we postpone the treatment of this step and reconsider it after the formal framework has been developed in the next section.

Let us assume that the above coarsement step has been verified. In order to verify the correctness of the original algorithm $\mathcal{S}$, we only have to verify the correctness of the specification $\mathcal{S}'$ with respect to the post-condition R. For this we need to show that the following predicate is an invariant of $\mathcal{S}'$:

$$I'\colon (\forall i, x \in V\colon \delta.i.x \leqslant D.i.x \leqslant N)$$

This invariant states that the distance $D.i.x$ of a node i to an arbitrary node x is between the minimum distance $\delta.i.x$ and N, $N > 1$. The invariant I' implies the invariant I for the algorithm $\mathcal{S}$ and is thus a stronger invariant than I.

At termination of $\mathcal{S}'$ the predicate T below holds:

$$T\colon (\forall i, x, j \in V\colon (i,j) \in E\colon D.i.j \leqslant 1 \wedge (D.i.x \leqslant D.j.x + 1 \vee x = i \vee x = j))$$

```
S' = [[ D.i.i := 0 for i ∈ V; D.i.j := N for i,j ∈ V(i ≠ j);
     do
[A'.i.j]  | D.i.j > 1 →
              D.i.j := 1
     for (i,j) ∈ E
[B'.i.j]  | D.i.x > D.j.x + 1 ∧ x ≠ i ∧ x ≠ j →
              D.i.x := D.j.x + 1
     for (i,j) ∈ E
     od
   ]|: D.i.j ∈ int for i,j ∈ V
```

Figure 3. The coarsed algorithm

We, thus, have to prove the following:

$$(T \wedge I') \Rightarrow R$$

Thereby, the correctness of the algorithm $\mathcal{S}'$ with respect to R is established and hence also the correctness of $\mathcal{S}$ with respect to R.

In this paper we are interested in the syntactic format of $\mathcal{S}'$, and therefore we consider this algorithm as the high level abstraction of $\mathcal{S}$. However, if we do not take the format into consideration, we can create a simpler abstract specification $\mathcal{S}''$ defined as:

$$\mathcal{S}'' = (D.i.x = dist(i, x) \textbf{ for } i, x \in V)$$

where

$$dist(i,i) \stackrel{\text{def}}{=} 0$$
$$dist(i,x) \stackrel{\text{def}}{=} min\{N, (dist(j, x) + 1 | (i,j) \in E)\}$$

The specification $\mathcal{S}''$ is not really a refinement of $\mathcal{S}'$. Rather $\mathcal{S}''$ defines how $\mathcal{S}'$ should be interpreted. According to $\mathcal{S}''$ the distances are calculated by a number of recursive calls. The returns from the calls when executing $\mathcal{S}''$ model how the actions in $\mathcal{S}'$ could be executed. Thus, we note that our technique spans a wide range of abstractions.

3. FORMALIZING THE COARSEMENT STEP

Let us now formalize the ideas presented in the previous section. Our goal is to be able to carry out the verification task in 3(iii) above. First of all we need a formal treatment of distributed systems. The approach of Morgan (1994) could be used in reverse, but it only applies for sequential programs. Another approach is the event-based formalism UNITY of Chandy and Misra (1988). UNITY has, however, no local variables and does not deal with program transformation, both of which are essential for our technique. Even though efforts have been made to include program transformations in UNITY (Udink, 1995), this formalism cannot be applied for our technique.

For the formal treatment of distributed systems we have chosen to use the action systems framework (Back and Kurki-Suonio, 1983), which is related to UNITY. The action system formalism is first very briefly outlined. Thereafter we give an informal description of the coarsement method within the action systems framework. Finally, the method is formalized as a program transformation rule within the refinement calculus for action systems.

In this section we focus on the coarsement step, step 3, of the reverse engineering method. However, formalizing this step makes it possible to work more methodologically also with steps 1 and 2. The action systems give us a very structured method for looking at a coarsement step by assisting in finding the target variable at each step and proposing a way to remove this variable.

3.1. Action systems

An *action system* $\mathcal{A}$ is a statement of the form

$$\mathcal{A} = |[\ \textbf{var}\ x := x_0;\ \textbf{do}\ A_1\ []\ \cdots\ []\ A_m\ \textbf{od}\ ]| : z$$

on *state variables* $y = x \cup z$, where the variables z are the *global* variables and x are the *local* variables. Each variable is associated with some domain of values. The set of possible assignments of values to the state variables constitutes the *state space*. The initialization statement $x := x_0$ assigns initial values to the state variables.

Each action A_i is of the form $g_i \rightarrow S_i$ where the *guard* g_i is a boolean expression on the state variables and the *body* S_i is a statement on the state variables. We denote the guard g_i of A_i by gA_i and the body S_i by sA_i. Furthermore, we say that an action is enabled in a state when its guard evaluates to *true* in that state.

The behaviour of an action system is that of Dijkstra's guarded iteration statement (Dijkstra, 1976) on the state variables: the initialization statement is executed first, thereafter, as long as there are enabled actions, one action at a time is non-deterministically chosen and executed. When all the actions are disabled the action system terminates. The exit condition of the action system $\mathcal{A}$ is, thus, $\neg(\bigvee gA_i)$.

If two actions are independent, i.e., they do not have any variables in common, they can be executed in parallel (Back and Sere, 1993). Their parallel execution is then equivalent to executing the actions one after the other, in either order.

The use of action systems requires an additional step in the reverse engineering method:

(0) Transform the distributed algorithm into an action system.

This step comes, however, naturally, since distributed algorithms can easily be seen as consisting of conditional operations that can be carried out a number of times in any order.

3.2. Informal description of the coarsement method

In step 1 of the reverse engineering method we identify the central data structures of a distributed algorithm. These correspond to the global variables z in action systems. The data structures identified in step 2 correspond to the local variables x. Step 3 is the coarsement step where data structures are abstracted away via modifications to the distributed algorithm. In terms of action systems, this means that some actions may be totally removed from the system and others are modified due to changes in the local data structures while the behaviour in terms of the remaining variables in the system is preserved.

Let $\mathcal{A}$ be the action system

$$\mathcal{A} = |[\ \textbf{var}\ x, x' := x_0, x_0';\ \textbf{do}\ A_1\ []\ \cdots\ []\ A_m\ []\ B_1\ []\ \cdots\ []\ B_n\ \textbf{od}\ ]| : z$$

Let us assume that we remove the variables x. This gives us the action system $\mathcal{A}'$

$$\mathcal{A}' = |[\ \textbf{var}\ x' := x_0';\ \textbf{do}\ A_1'\ []\ \cdots\ []\ A_m'\ \textbf{od}\ ]| : z$$

which has the same global variables z as $\mathcal{A}$ has. The action system $\mathcal{A}$ has the local variables x, x', whereas $\mathcal{A}'$ only has the local variables x' left. This change in the local variables is reflected in the actions. For each action A_i in $\mathcal{A}$ there is a corresponding *changed* action A_i' in $\mathcal{A}'$ while the actions B_j in $\mathcal{A}$ are *removed* and do not correspond to any actions in $\mathcal{A}'$.

Let $R(x, x', z)$ be a relation on the variables x, x', z in $\mathcal{A}$. We say that the action system $\mathcal{A}'$ is a *(correct) coarsement* of the action system $\mathcal{A}$ using the *abstraction relation* (or *data invariant*) $R(x, x', z)$, denoted $\mathcal{A} \sqsupseteq_R \mathcal{A}'$, if the following conditions are satisfied:

(1) *Initialization*: the initialization $x, x' := x_0, x_0'$ will establish $R(x, x', z)$.
(2) *Changed actions*:
   (a) each changed action A_i' has the same effect on the variables x', z as the corresponding action A_i when $R(x, x', z)$ holds,
   (b) each action A_i will establish $R(x, x', z)$, and
   (c) the guard of each changed action A_i' is implied by the guard of the corresponding action A_i when $R(x, x', z)$ holds.
(3) *Removed actions:*
   (a) none of the removed actions B_j has any effect on the variables x', z when $R(x, x', z)$ holds, and
   (b) each action B_j establishes $R(x, x', z)$.
(4) *Termination of removed actions*: executing only actions that are to be abstracted away in $\mathcal{A}$ in an initial state where $R(x, x', z)$ holds will necessarily terminate.
(5) *Exit condition*: all actions in the action system $\mathcal{A}'$ are disabled whenever all actions in the action system $\mathcal{A}$ are disabled when $R(x, x', z)$ holds, i.e., the exit condition $\neg g\mathcal{A}'$ of $\mathcal{A}'$ is implied by the exit condition $\neg(g\mathcal{A} \vee g\mathcal{B})$ of $\mathcal{A}$ when $R(x, x', z)$ holds.

The conditions (1)–(5) above guarantee that the result of the computation of $\mathcal{A}$ in terms of the state variables x', z is preserved in $\mathcal{A}'$ when the action systems are considered in isolation. When the action system occurs in some environment interacting with other action systems via the global variables z, these requirements are not sufficient. To take the environment into account, we have to add one more condition which states that the environment preserves the relation R. (The formal treatment of the environment is outside the scope of the current paper. The interested reader is referred to (Back and Sere, 1993).)

3.3. Formalizing coarsement

When taking a closer look at the coarsement method above, we observe that we have been applying, in a reverse direction, a program modularization and structuring method called *superposition* (Francez and Forman, 1990; Katz, 1993; Back and Kurki-Suonio, 1983). In superposition some new functionality is added into an algorithm in the form of new added variables and assignments to these while the original computation is preserved, i.e., exactly the converse to the way coarsements are carried out here.

The superposition method has been formalized as a program refinement rule within the refinement calculus (Back and Sere, 1993) following the formalization in Back (1978). Let us now briefly describe this calculus.

Let S be a statement on the program variables x, z and S' a statement on the program

variables x', z. Let $R(x, x', z)$ be a relation on these variables. Then S is *data refined* by S' using the abstraction relation R, denoted $S \leqslant_R S'$, if for any postcondition Q we have that

$$R \wedge \text{wp}(S, Q) \Rightarrow \text{wp}(S', \exists x{:}R \wedge Q)$$

where wp is the *weakest precondition* predicate transformer (Dijkstra, 1976). Intuitively, a statement S is data refined by a statement S' using R, if providing that R holds (i) whenever S is guaranteed to terminate, S' is also guaranteed to terminate, and (ii) any possible outcome of S' for some initial state is also a possible outcome of S for this same initial state. This means that a refinement may either extend the domain of termination of a statement or decrease the non-determinism of the statement, or both. Successive refinements are modelled as follows: If $S_0 \leqslant_{R_1} S_1$ and $S_1 \leqslant_{R_2} S_2$ then $S_0 \leqslant_{R_1 \wedge R_2} S_2$.

Data refinements of actions is defined in a similar way. Let A be an action on the program variables x, z and A' an action on the program variables x', z. Let $R(x, x', z)$ be a relation on these variables. Then A is data refined by A' using R, denoted $A \leqslant_R A'$, if

(i) $\{gA'\}$; $sA \leqslant_R sA'$ and
(ii) $R \wedge gA' \Rightarrow gA$.

Intuitively, (i) means that A' has the same effect on the program variables that A has when R holds and moreover, A' establishes R. The condition (ii) requires that A is enabled whenever A' is enabled provided R holds.

The superposition refinement of action systems is a special kind of data refinement and it is formally expressed as follows (Back and Sere, 1993).

Let $\mathcal{C}$ and $\mathcal{C}'$ be the two action systems:

$$\mathcal{C} = |[\ \textbf{var}\ y := y_0;\ \textbf{do}\ C_1\ [] \ \ldots\ []\ C_m\ \textbf{od}\ ]| : z \text{ and}$$
$$\mathcal{C}' = |[\ \textbf{var}\ y,y' := y_0,y_0';\ \textbf{do}\ C_1'\ [] \ \ldots\ []\ C_m'\ []\ D_1\ []\ \ldots\ []\ D_n\ \textbf{od}\ ]| : z$$

Let $g\mathcal{C}$ be the disjunction of the guards of the C_i actions, $g\mathcal{C}'$ the disjunctions of the guards of the C_i' actions and $g\mathcal{D}$ the disjunction of the guards of the D_j actions. Then $\mathcal{C} \leqslant_R \mathcal{C}'$ using $R(y, y', z)$, if

(1) $R(y_0, y_0', z)$,
(2) $C_i \leqslant_R C_i'$, for $i = 1, \ldots, m$,
(3) $skip \leqslant_R D_j$, for $j = 1, \ldots, n$,
(4) $R \Rightarrow \text{wp}(\textbf{do}\ D_1\ [] \ \ldots\ []\ D_n\ \textbf{od}, true)$,
(5) $R \wedge \neg(g\mathcal{C}' \vee g\mathcal{D}) \Rightarrow \neg g\mathcal{C}$.

Hence, following the superposition rule, an action system $\mathcal{C}$ is correctly data refined by another action system $\mathcal{C}'$ using the abstraction relation R when

(1) the initialization in $\mathcal{C}'$ establishes R,
(2) every action C_i is data refined by the corresponding action C_i' using R,
(3) every action D_j is a data refinement of the empty statement *skip* using R,

(4) the computation denoted by the actions $D_1, \ldots, D_n$ terminates provided R holds, and
(5) the exit condition of $\mathcal{C}'$ implies that of $\mathcal{C}$ when R holds.

When looking at the superposition rule in a reverse direction we get the conditions for a correct coarsement as given in the previous section. Therefore, we simply have that if

$$\mathcal{C} \leqslant_R \mathcal{C}'$$

for action systems $\mathcal{C}$ and $\mathcal{C}'$ and abstraction relation R, then

$$\mathcal{C}' \sqsupseteq_R \mathcal{C}$$

Thus, coarsement is superposition refinement in reverse. By coarsement we may then either diminish the domain of termination of a statement or increase the non-determinism of a statement, or both.

3.4. Verification of a coarsement step

As an example of the coarsement formalization we now verify the correctness of the coarsement step from $\mathcal{S}$ to $\mathcal{S}'$ using the informal argumentation in Section 3.2. This is step 3(iii) in the reverse engineering method.

We show that $\mathcal{S} \sqsupseteq_{I \wedge P} \mathcal{S}'$ where I and P are given in Section 2, but we repeat them below for convenience.

$$I\colon (\forall i, x \in V\colon 0 \leqslant D.i.x \leqslant N)$$

$$P\colon P_1 \wedge P_2 \wedge P_3$$
$$P_1\colon (\forall i, j, x \in V, l \in \mathit{int}\colon (i,j) \in E, x \neq j\colon$$
$$\quad <x,l> \in Q.j.i \Rightarrow (0 \leqslant l \leqslant N) \wedge (D.j.x \leqslant l))$$
$$P_2\colon (\forall i, j, x \in V, l \in \mathit{int}\colon (i,j) \in E, x \neq i, x \neq j\colon$$
$$\quad (D.j.x = l \wedge l < N) \Rightarrow (\exists k \in \mathit{int}\colon Q.j.i.k = <x,l>))$$
$$P_3\colon (\forall i, j, x \in V\colon (i,j) \in E, x \neq i, x \neq j\colon$$
$$\quad (\exists l \in \mathit{int}\colon <x,l> \in Q.j.i \wedge D.i.x > l + 1) \equiv D.i.x > D.j.x + 1)$$

Following the formalization above we need to show the correctness of the five conditions for a correct coarsement, where D is the remaining variable and Q is the data structure that is abstracted away:

(1) *Initialization*
  (a) The initialization in the coarsed algorithm $\mathcal{S}'$ has the same effect on the variable D as the initialization in the original algorithm $\mathcal{S}$, because the assignment to the removed variable Q is the only assignment that was abstracted away.
  (b) The initialization in the original algorithm $\mathcal{S}$ will establish the data invariant $I \wedge P$. This is seen as the distances are set to zero or N and the links are initially empty.

(2) *Changed actions*

The changed actions in $\mathcal{S}$ are $A.i.j$ and $B.i.j$.

(a) Hence, we have to show that the bodies of the actions, $A'.i.j$ and $B'.i.j$, have the same effect on the remaining variables as the actions, $A.i.j$ and $B.i.j$, respectively. The actions $A.i.j$ and $A'.i.j$ are immediately seen to have this property, because the assignments to D are unchanged. In the actions $B.i.j$ we receive a distance, denoted by l, over the channel $Q.j.i$. Owing to P this distance is clearly some distance value $D.j.x$ and hence, the effect on D remains unchanged.

(b) The actions $A.i.j$ establish the data invariant, because the only changed distance $D.i.j$ from node i to neighbour j is given the value 1. This establishes I. These distances of length 1 are also sent on the links to the neighbouring nodes establishing P. Furthermore, the actions $B.i.j$ also establish the data invariant. A new value to $D.i.x$ in node i is assigned upon receiving a new shorter distance to destination x establishing I, and this new distance is sent to all the neighbours of node i establishing P.

(c) The guards of the old actions should imply the guards of the corresponding changed actions when the data invariant holds. This is true since the guards of the actions $A.i.j$ and $A'.i.j$ are identical. Moreover, the guard of $B.i.j$ implies the guard of $B'.i.j$ when P_3 holds.

(3) *Removed actions*

The actions $C.i.j$ in $\mathcal{S}$ are removed.

(a) The action $C.i.j$ has no effect on the remaining variable D when the data invariant $I \wedge P$ holds, since this action does not assign to D.

(b) The removed action $C.i.j$ preserves the data invariant P due to the following reasoning. The invariants P_1 and P_2 are preserved, because these invariants only state what is sent on the links $Q.j.i$. In $C.i.j$ we do not send any new messages on the links and receiving an element from a link $Q.j.i$ does not affect these invariants. If the invariant P_3 holds when the action $C.i.j$ is enabled it also holds after $C.i.j$ is executed. This is true, since

$$(x = i \vee (x \neq i \wedge x \neq j \wedge \neg(D.i.x > l + 1)))$$

holds when the action is enabled. In the case $x = i$ *the invariant* P_3 *holds trivially. On the other hand, if* $\neg(D.i.x > l + 1)$, removing the message $<x,l>$ from the link $Q.j.i$ does not falsify the invariant P_3, since P_3 is only concerned with messages $<x,l>$ where the distance l is less than $D.i.x$.

(4) *Termination of removed actions*

In the removed action $C.i.j$, messages are removed from the link $Q.j.i$. There will only be a finite number of messages on each link and the action removes one message at a time. Finally, the link $Q.j.i$ will contain no messages and the removed action will not be enabled. Hence, the loop containing only the removed actions will necessarily terminate.

5) *Exit condition*

The exit condition of the action system $\mathcal{S}'$ implies the exit condition of the action system $\mathcal{S}$ when the data invariant holds. We show using contrapositive reasoning

$I \wedge P \wedge$

$$
\begin{aligned}
&((\exists i, j \in V: (i,j) \in E: D.i.j > 1) \\
&\quad \vee (\exists i, j, x \in V: (i,j) \in E, x \neq i, x \neq j: D.i.x > D.j.x + 1)) \\
&\Rightarrow \\
&(\exists i, j \in V: (i,j) \in E: D.i.j > 1) \\
&\quad \vee (\exists i, j, x \in V: (i,j) \in E: \\
&\quad Q.j.i \neq <> \wedge \mathrm{hd}(Q.j.i) = <x,l> \wedge x \neq i \wedge x \neq j \wedge D.i.x > l + 1) \\
&\quad \vee (\exists i, j, x \in V: (i,j) \in E: \\
&\quad Q.j.i. \neq <> \wedge \mathrm{hd}(Q.j.i) = <x,l> \wedge \\
&\quad ((x = i \vee (x \neq i \wedge x \neq j \wedge \neg(D.i.x > l + 1)))
\end{aligned}
$$

If

$$
\begin{aligned}
&(\exists i, j \in V: (i,j) \in E: D.i.j > 1) \\
&\quad \vee (\exists i, j, x \in V: (i,j) \in E, x \neq i, x \neq j: D.i.x > D.j.x + 1)
\end{aligned}
$$

is false or $D.i.j > 1$ is true for some $(i,j) \in E$, then the implication holds trivially. On the other hand if

$$(\exists i, j, x \in V: (i,j) \in E, x \neq i, x \neq j: D.i.x > D.j.x + 1)$$

is true for a given pair (j, x) and $D.i.j > 1$ is false, we have two cases. In the case $(x = i \wedge x \neq j)$ the antecedent of the implication is false and the whole condition is true. For the case $(x \neq i \wedge x \neq j)$ we can rely on the invariant P_3. Thus, we have that for these cases the exit condition of the original system $\mathcal{S}$ implies that of the coarsed system $\mathcal{S}'$ when $I \wedge P$ holds.

Hence, we have verified the correctness of our coarsement step. (Totally formal argumentation is applied in Sere and Waldén (1994).)

4. REVERSE ENGINEERING A DISTRIBUTED ALGORITHM

In a distributed network the basic problem is the routeing of messages. That is, if a node in the network wants to send a message to some other node in the network or receives a message destined for some other node, a method is needed to enable the node to decide over which outgoing link it has to send the message. Algorithms solving this problem are called routeing algorithms. Our previous example was a simple minimum-hop routeing algorithm. We now analyse a more elaborate distributed algorithm for minimum-hop route maintenance as a case study on the reverse engineering method.

Desirable properties of routeing algorithms are for example correctness, optimality and robustness. Correctness seems easy to achieve in a static network, but is far less trivial in case communication links go down and come up as they can do in practice. Optimality is concerned with finding the 'quickest' routes. Ideally, a route should be chosen for a message on which it will encounter the least delay, but as this depends on the amount of traffic on the way, such routes are hard to predict and hence the goal is actually difficult to achieve. A frequent compromise is to minimize the number of hops, i.e., the number of links over which the message travels from origin node to destination. Robustness is concerned with the ease with which the routeing scheme is adapted in case of topological changes.

In this work we consider the minimum-hop routeing algorithm of Tajibnapis (1977). This is a non-trivial distributed algorithm that is of great interest and that has been a subject of several studies, see for instance Lamport (1982); Schoone (1991). It has not, however, been derived in any formal or stepwise manner. Hence, the algorithm of Tajibnapis is still worth studying.

Reverse engineering the algorithm of Tajibnapis creates a high level specification of it. After the coarsement steps the algorithm is easier to understand and reason about than the original algorithm, because it can now be seen as consisting of a number of layers interacting with each other. Furthermore, each coarsement step can be formally verified. By giving a proof of correctness for the high level specification we can obtain a proof of correctness of the entire algorithm.

The algorithm of Tajibnapis

The algorithm of Tajibnapis is in Figure 4 given as an action system. The algorithm works in a dynamic environment, where communication links beween nodes can come up and go down. Hence, the minimum-hop distance between two nodes can change in time. If the link situation is stable, i.e., links stop coming up and going down, then all the computed distances between pairs of nodes will be the minimum distances.

We will now analyse the algorithm following the reverse engineering method adapting the informal reasoning of the algorithm given by Tajibnapis to our terms. We start by identifying the data structures as required by steps 1 and 2 of the method.

(1) As in the example algorithm in Section 2, the central data structure for the algorithm of Tajibnapis is the array D, which at termination will for all the nodes i have the minimum distances $\delta.i.x$ to every other node x. This is expressed in the postcondition R:

$$R\colon (\forall i, x \in V\colon D.i.x = \delta.i.x)$$

(2) Let us now take a closer look at the other data structures in the algorithm of Tajibnapis. At any point of time, each node i knows the identities of its neighbouring nodes, *nbrs.i*, and the total number of nodes N. The links between the nodes are modelled by the variable Q so that $Q.i.j$ denotes the link from node i to node j. The distance computation proceeds by the nodes sending distance messages on these links. The messages consist of two parts, $<x,l>$, where x is the destination node and l is the distance from the sending node to x. Besides these data messages, certain control messages marked *up* and *down* might appear on the links. These messages model link repairs and failures in the system. They are inserted into the links by some environment that is not specified here. When a link between the nodes i and j comes up or goes down, the environment sends a control message to both i and j, on $Q.i.j$ and $Q.j.i$, simultaneously. At initialization all links are assumed to be empty. The appearance of an *up*-message on a link will establish the link and data messages can be sent along it. A *down*-message denotes that the link is not available. A link $Q.i.j$ has a pointer $p.i.j$ that points at the next message to be read from the link. This variable is considered to be a so called history

```
T =  |[ var Q.i.j value list  for i,j ∈ V; p.i.j ∈ int  for i,j ∈ V;
        nbrs.i index set  for i ∈ V;
        dsn.i.j ∈ V  for i,j ∈ V; Dtab.i.x.j ∈ int  for i,j,x ∈ V;

     D.i.i := 0  for i ∈ V; D.i.j := N  for i,j ∈ V(i ≠ j);
     Q.i.j :=<>  for i,j ∈ V(i ≠ j); p.i.j := -1  for i,j ∈ V(i ≠ j)
     dsn.i.j := none  for i,j ∈ V(i ≠ j); nbrs.i := ∅  for i ∈ V;          (1)
     Dtab.i.x.j := ∞  for i,j,x ∈ V(i ≠ j, x ≠ i);

     do
[A.i.j]  | Q.j.i.q ≠<> ∧ Q.j.i.q =< up >→
           p.j.i := p.j.i + 1; nbrs.i := nbrs.i ∪ {j};
           dsn.i.j := j; Dtab.i.j.j := 0; D.i.j := 1;                     (2)
           Q.i.a := Q.i.a, < j, 1 >  for a ∈ nbrs.i, a ≠ j;
           (Dtab.i.x.j := N; if |nbrs.i| = 1 → dsn.i.x := j fi;           (3)
            Q.i.j := Q.i.j, < x, D.i.x >
           )  for x ∈ V, x ≠ i ∧ x ≠ j
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[B.i.j]  | Q.j.i.q ≠<> ∧ Q.j.i.q =< down >→
           p.j.i := p.j.i + 1; nbrs.i := nbrs.i - {j};
           (Dtab.i.x.j := ∞;
            if nbrs.i ≠ ∅ ∧ dsn.i.x = j →                                 (4)
               olddist := D.i.x;
               dsn.i.x := d'.(d' ∈ nbrs.i ∧
                              Dtab.i.x.d' = min_{a∈nbrs.i}{Dtab.i.x.a});  (5)
               D.i.x := min{N, 1 + Dtab.i.x.(dsn.i.x)};                   (6)
               if olddist ≠ D.i.x → Q.i.a := Q.i.a, < x, D.i.x >  for a ∈ nbrs.i fi
            elseif nbrs.i = ∅ → D.i.x := N; dsn.i.x := none               (7)
            fi
           )  for x ∈ V, x ≠ i
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[C.i.j]  | Q.j.i.q ≠<> ∧ Q.j.i.q ≠< up > ∧ Q.j.i.q ≠< down >→
           p.j.i := p.j.i + 1; < x, l >:= Q.j.i.(p.j.i);
           if x ≠ i ∧ j ∈ nbrs.i →
              Dtab.i.x.j := l;
              if dsn.i.x = j ∨ D.i.x > l + 1 →                             (8)
                 olddist := D.i.x;
                 dsn.i.x := d'.(d' ∈ nbrs.i ∧
                                Dtab.i.x.d' = min_{a∈nbrs.i}{Dtab.i.x.a}); (9)
                 D.i.x := min{N, 1 + Dtab.i.x.(dsn.i.x)};                 (10)
                 if D.i.x ≠ olddist → Q.i.a := Q.i.a, < x, D.i.x >  for a ∈ nbrs.i fi
              fi
           fi
     for i,j ∈ V(i ≠ j), q = p.j.i + 1
     od
  ]|: D.i.j ∈ int  for i,j ∈ V
```

Figure 4. The minimum-hop route maintenance algorithm of Tajibnapis

variable (Lamport, 1982) and it does not belong to the algorithm. It is only needed for verification, but for completeness we have included it in the program.

The algorithm consists of the actions *A.i.j*, *B.i.j* and *C.i.j*. The action *A.i.j* establishes links between nodes by adding a new neighbour j to the *nbrs.i* set. It is enabled whenever there is an *up*-message on some link *Q.j.i* to node i. Whenever there is a *down*-message on some link *Q.j.i* to node i the action *B.i.j* will be enabled for some node j. The effect of this action is that node j will be removed from the neighbour set of node i. Moreover, distances *D.i.x* to every other node x are recomputed. Upon receiving a data message $<x,l>$ from a neighbour j in action *C.i.j* node i recomputes the distance *D.i.x* as the minimum of the old value of *D.i.x* and the received distance l plus one.

In order to be able to recompute a new estimate for the minimum distance, each node stores the information it receives from its neighbours in a table *Dtab*. An element *Dtab.i.x.j* in node i contains for every destination $x \neq i$ and for every (current) neighbour j of node i, the most recent distance information l received from node j in a message $<x,l>$ along *Q.j.i* in action *C.i.j*.

The variable *dsn.i.x contains the identity of that neighbour via which node i has the shortest path to node x, the so called downstream* neighbour of node i for destination x. It is initially set to *none* for every node and it is updated in every action. A node has no downstream neighbour if and only if it has no neighbours.

As described by Tajibnapis (1977) the algorithm assumes that the total number of nodes in the network or, at least an upper bound, N, of it is known by all the nodes. This number is needed in case the network becomes disconnected. The distances between some nodes will then be infinite. As the algorithm tends to increase the distance estimates with one hop at a time, we need a way to 'assume' that the distance is infinity to prevent the program ever terminating. The underlying observation is that if the total number of nodes is bounded by N, the longest possible finite distance between nodes is bounded by $N - 1$. Both the symbol ∞ and the symbol N are used for infinite distances. The symbol ∞ is only used for *Dtab.i.x.j* when the link from i to j is down. It is required that the total number of nodes is at least two, i.e., $N > 1$. Hence, we have the following invariant in the system:

$$U_0\text{: } \forall i, j, x \in V, l \in \text{int: } i \neq j, x \neq i, x \neq j\text{:}$$
$$(j \in nbrs.i \Rightarrow 0 \leq Dtab.i.x.j \leq N) \wedge (dsn.i.x = none \Leftrightarrow nbrs.i = \emptyset)$$
$$\wedge (0 \leq D.i.x \leq N) \wedge (<x,l> \in Q.i.j \Rightarrow 0 < l \leq N)$$

5. COARSEMENT STEPS

In the previous section we identified the different data structures of the algorithm of Tajibnapis. We will now apply two coarsement steps on this algorithm. Hence, we continue our work with the coarsement step, step 3, of the reverse engineering method. In the first coarsement step we abstract away the variable *dsn* that contains direction information for routeing messages. In the second coarsement step we abstract away the variable *Dtab* with knowledge about the values in the distance tables D of the neighbours for each node.

The only variables that remain after these two coarsements are the distances D to every

destination, the neighbour set *nbrs* and the links Q. The other variables in the algorithm of Tajibnapis are expressed using these remaining variables. In this section we concentrate on the first two parts 3(i) and 3(ii) of the coarsement step. The verification step 3(iii) is reported in the full paper (Sere and Waldén, 1994).

5.1. Coarsement step 1: removing direction information

We start from the algorithm $\mathcal{T}$ of Tajibnapis in Figure 4, where the information about the downstream neighbour of node i for routeing messages to the destination node x is stored in the variable *dsn.i.x*. This variable can, however, be abstracted away, because its information content is also available via the *Dtab*-variable. All the computation that involves the *dsn*-variable is in Figure 4 marked with `this font`. Furthermore, every line where the modifcation will take place is indicated by a label in the figure.

(i) Initially the variable *dsn.i.j* has the value *none*. At this point *Dtab.i.x.j* has value ∞ for every pair of nodes i, j and nodes $x \neq i$. Hence, we have the following correspondence:

$$U_1\colon \forall i, j \in V\colon i \neq j\colon dsn.i.j = none \Rightarrow (\forall x \in V\colon x \neq i\colon Dtab.i.x.j = \infty)$$

This is also an invariant of the entire computation as can be easily proved from the program text of $\mathcal{T}$.

The variable *dsn* is assigned new values in all three actions. The action *A.i.j* is a sort of initialization for the computation. Here a node j will become a new neighbouring node of node i. This is reflected in *dsn.i.j* which is assigned the value j. Also *Dtab.i.j.j* is updated and assigned value 0. Therefore,

$$U_2\colon \forall i, j \in V\colon i \neq j\colon dsn.i.j = j \equiv Dtab.i.j.j = 0$$

which is an invariant of the system. We also have that

$$U_3\colon \forall i, j, x \in V\colon i \neq j\colon dsn.i.x = j \Rightarrow j \in nbrs.i$$

i.e., the shortest path can only go via neighbours. If node j happens to be the only neighbour of node i, the variable *dsn.i.x* gets the value j:

$$U_4\colon \forall i, j \in V\colon i \neq j\colon |nbrs.i| = 1 \wedge j \in nbrs.i \Rightarrow (\forall x \in V\colon x \neq i\colon dsn.i.x = j)$$

The purpose of action *B.i.j* is to change the neighbourhood of node i by removing node j. Now the value of *Dtab.i.x.j* is set to ∞ for every node x such that $x \neq i$. We have two cases: if the neighbourhood becomes empty, *dsn.i.x* is assigned *none* for every node x, $x \neq i$. This is in accordance with the invariant U_1 above. If there are neighbours left, we also have two cases: either $dsn.i.x = j$ or $dsn.i.x \neq j$ for every x, $x \neq i$. If $dsn.i.x \neq j$ the value of *dsn.i.x* need not be changed, because the shortest path to x is not via j. If $dsn.i.x = j$, this value must be updated, as $j \notin nbrs.i$ violates the invariant U_3. The update respects the correspondence

$$U_5\colon \forall i, a, x \in V\colon i \neq a, i \neq x\colon dsn.i.x = a \Rightarrow$$

$$Dtab.i.x.a. = min\{Dtab.i.x.b \mid b \in nbrs.i\}$$

Moreover, we have that

$$\forall i, a, x \in V: i \neq a, i \neq x: dsn.i.x = a \Rightarrow D.i.x = min\{N, Dtab.i.x.a + 1\}$$

This gives us the following invariant:

$$U_6: \forall i, a, x \in V: i \neq a, i \neq x: dsn.i.x = a \Rightarrow (\exists k \in V: k \in nbrs.i: D.i.x = Dtab.i.x.k + 1 \vee D.i.x = Dtab.i.x.k = N)$$

In the action *C.i.j* the value *dsn.i.x* in node *i* may change due to the received update message $<x,l>$ from node *j*. Again we have several cases: if $x = i$ or $j \notin nbrs.i$ this message cannot affect the value of *dsn.i.x*. However, in the case of $x \neq i \wedge j \in nbrs.i$ we have received a possibly new shortest path information from node *j* to node *x*. Therefore *Dtab.i.x.j* is set to *l*. Now we get some more cases: if $dsn.i.x \neq j \wedge D.i.x \leq l + 1$, we need not update *dsn.i.x*, because the shortest path to *x* remains via some node different from *j*. If on the other hand $dsn.i.x = j \vee D.i.x > l + 1$, we need to recompute *dsn.i.x*. In this case the update follows the same pattern as above, i.e., $U_5 \wedge U_6$ is an invariant of the system. Thus, the invariant for the variable *dsn* is the following:

$$U: U_0 \wedge U_1 \wedge U_2 \wedge U_3 \wedge U_4 \wedge U_5 \wedge U_6$$

(ii) When we remove the *dsn*-variable from the algorithm, we rely on our invariants. The new algorithm $\mathcal{T}_1$ is given in Figure 5. The initialization and the action *A.i.j* are straightforward to modify: due to invariant U_1 the assignment of *none* to the variable *dsn* can be removed (change (1)) in the initialization. In the action *A.i.j* the assignments $dsn.i.j := j$ and $dsn.i.x := j$ can be removed (changes (2) and (3)) relying on the invariants U_2 and U_3, respectively. Besides, by the invariant U_4 the *dsn*-variable can be removed from *A.i.j* in the case where $|nbrs.i| = 1$ (change (3)).

Let us consider the actions *B.i.j* and *C.i.j*. In *B.i.j* we rely on the invariant U_1 when removing the reference to *dsn.i.x* when *nbrs.i* becomes empty (change (7)). The invariants U_5 and U_6 allow us to carry out the necessary modifications which remove the other references to the variable *dsn* (changes (4)–(6) and (8)–(10)).

Here some care must be taken when the guards

$$(nbrs.i \neq \emptyset \wedge dsn.i.x = j) \text{ and } (dsn.i.x = j \vee D.i.x > l + 1)$$

of the **if** statements are modified (changes (4) and (8)) as the *dsn*-variable appears in these. We have from U_6 that

$$dsn.i.x = j \Rightarrow (\exists k \in V: k \in nbrs.i: D.i.x = Dtab.i.x.k + 1 \vee D.i.x = Dtab.i.x.k = N)$$

```
T1 = |[ var Q.i.j value list for i,j ∈ V; p.i.j ∈ int for i,j ∈ V;
        nbrs.i index set for i ∈ V;
        Dtab.i.x.j ∈ int for i,j,x ∈ V;

     D.i.i := 0 for i ∈ V; D.i.j := N for i,j ∈ V(i ≠ j);
     Q.i.j :=<> for i,j ∈ V(i ≠ j); p.i.j := -1 for i,j ∈ V(i ≠ j);
     nbrs.i := ∅ for i ∈ V;
     Dtab.i.x.j := ∞ for i,j,x ∈ V(i ≠ j, x ≠ i);                    (a)

     do
[A.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q = < up > →
          p.j.i := p.j.i + 1; nbrs.i := nbrs.i ∪ {j};
          Dtab.i.j.j := 0; D.i.j := 1;                               (b)
          Q.i.a := Q.i.a, < j, 1 > for a ∈ nbrs.i, a ≠ j;
          (Dtab.i.x.j := N; Q.i.j := Q.i.j, < x, D.i.x >             (c)
          ) for x ∈ V, x ≠ i ∧ x ≠ j
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[B.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q = < down > →
          p.j.i := p.j.i + 1; nbrs.i := nbrs.i - {j};
          (Dtab.i.x.j := ∞;                                          (d)
           if nbrs.i ≠ ∅ ∧ (∀k : k ∈ nbrs.i : D.i.x ≠ Dtab.i.x.k + 1) →   (e)
              olddist := D.i.x; D.i.x := min_{a∈nbrs.i}{N, Dtab.i.x.a + 1};   (f)
              if olddist ≠ D.i.x → Q.i.a := Q.i.a, < x, D.i.x > for a ∈ nbrs.i fi
           elseif nbrs.i = ∅ → D.i.x := N
           fi
          ) for x ∈ V, x ≠ i
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[C.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q ≠ < up > ∧ Q.j.i.q ≠ < down > →
          p.j.i := p.j.i + 1; < x, l > := Q.j.i.(p.j.i);
          if x ≠ i ∧ j ∈ nbrs.i →
             Dtab.i.x.j := 1;                                        (g)
             if (∀k : k ∈ nbrs.i : D.i.x ≠ Dtab.i.x.k + 1) ∨ D.i.x > l + 1 →   (h)
                olddist := D.i.x; D.i.x := min_{a∈nbrs.i}{N, Dtab.i.x.a + 1};   (i)
                if D.i.x ≠ olddist → Q.i.a := Q.i.a, < x, D.i.x > for a ∈ nbrs.i fi
             fi
          fi
     for i,j ∈ V(i ≠ j), q = p.j.i + 1
     od
]|: D.i.j ∈ int for i,j ∈ V
```

Figure 5. The algorithm after abstracting away direction information

holds immediately before the execution of these actions. In both actions the assignments to *Dtab.i.x.j* destroy this invariant. We have, however, that

$$dsn.i.x = j \equiv (\forall k \in V\colon k \in nbrs.i\colon D.i.x \neq Dtab.i.x.k + 1)$$

holds in the algorithm after the assignments $Dtab.i.x.j := \infty$ in *B.i.j* and $Dtab.i.x.j := l$ in *C.i.j*, which explains the modification of the guards to

$$nbrs.i \neq \emptyset \wedge (\forall k\colon k \in nbrs.i\colon D.i.x \neq Dtab.i.x.k + 1)$$

and

$$(\forall k\colon k \in nbrs.i\colon D.i.x \neq Dtab.i.x.k + 1) \vee D.i.x. > l + 1)$$

respectively.

The effect of this step is that non-determinism has been added. In the original algorithm $\mathcal{T}$ when a *down* message appears on a link *Q.j.i* for node *i* we know which distances must be recomputed, namely those that are reachable via node *j* as the downstream neighbour. In $\mathcal{T}_1$ every *down*-message may cause a complete recomputation of the distance table. Also in the case when a data message <*x*,*l*> appears on a link *Q.j.i* for a node *i* the computation in $\mathcal{T}_1$ is more non-deterministic, as there is no knowledge of the downstream neighbour left anymore.

5.2. Coarsement step 2: removing knowledge about neighbour distances

As the second coarsement step we abstract away the knowledge a node has about the distances of its neighbours to all the other nodes, i.e., the variable *Dtab*. The coarsement step is performed on the algorithm $\mathcal{T}_1$ in Figure 5, where all the computation that involves this variable is denoted with `this font`. Furthermore, every line where the modification on $\mathcal{T}_1$ will take place is indicated by a label in the figure.

(i) The invariant W_1 below gives the properties for the variable *Dtab.i.x.j* at initialization. The value of *Dtab.i.x.j* is ∞, if node *j* is not among the neighbours of node *i*:

$$W_1\colon \forall i, j \in V\colon i \neq j\colon j \notin nbrs.i \Rightarrow (\forall x \in V\colon x \neq i\colon Dtab.i.x.j = \infty)$$

This can be easily shown to be an invariant of the system.

If node *j* in action *A.i.j* is joining the neighbourhood of node *i*, the variable *Dtab.i.x.j* is set to contain the value 0 for $x = j$:

$$W_2\colon \forall i, j \in V\colon j \in nbrs.i \Rightarrow Dtab.i.j.j = 0$$

For $x \neq j$ it is assigned the value *N* (this case is formalized in W_3 below).

Let us consider action *B.i.j*. In this action, node *j* is removed from the set *nbrs.i*. Hence, the invariant W_1 gives the required information about the update of *Dtab.i.x.j* for every *x*. We also now have to recompute the *D.i.x* values, as they depend on *Dtab.i.x.j*. The information stored in *Dtab.i.x.j* in node *i* is received from node *j* via the link *Q.j.i*. These *Dtab*-values can therefore be found in *Q.j.i* as follows from the invariant

$$W_3\colon \forall i, j, x \in V : j \in nbrs.i, x \neq i, x \neq j\colon \\ Dtab.i.x.j = N \vee (\exists r \in int : H(j,i,r,x) \Rightarrow Dtab.i.x.j = snd(Q.j.i.r))$$

where $H(j,i,r,x)$ is given below:

$$H(j,i,r,x) = (\exists l \in int: 0 \leq r \leq p.j.i \wedge Q.j.i.r = <x,l> \wedge (\forall t, d \in int: r < t \leq p.j.i: Q.j.i.t \neq <x,d> \wedge Q.j.i.t \neq <up> \wedge Q.j.i.t \neq <down>))$$

This predicate states that the value of *Dtab.i.x.j* for a particular node x is found at place r in *Q.j.i*, where r is pointing to the first data message with destination x that precedes a control message when we inspect *Q.j.i* starting from the tail, i.e., it is the most recently sent data message from j to i for destination x after the latest control message on this link.

The actions *C.i.j* update *Dtab.i.x.j* upon receiving a message $<x,l>$ from *Q.j.i*. Hence, this value is available via *Q*. The *D.i.x* update proceeds as before following the invariant W_3 above. Hence, the invariant W describes the *Dtab* variable:

$$W: W_1 \wedge W_2 \wedge W_3$$

(ii) We now rely on our invariants when doing modifications to the code. Here the changes to the initialization and the action *A.i.j* are straightforward. In the initialization the assignment of ∞ to the variable *Dtab* can be removed (change (a)) due to the invariant W_1, which gives the value of *Dtab.i.x.j* when j is not a neighbour of i. The assignments *Dtab.i.j.j* := *0* and *Dtab.i.x.j* := *N* can be removed in *A.i.j* (changes (b) and (c)) relying on the invariants W_2 and W_3, respectively.

When modifying action *B.i.j* we rely on invariants W_1 and W_3, where W_1 is needed as node j is not a neighbour of node i anymore (change (d)). In this action the new *D.i.x*-value is computed from the distances received earlier from the neighbours, i.e., from the *Dtab*-variable. This value can instead be computed from the data messages on link *Q.j.i*. According to W_3 we can substitute snd(*Q.j.i.r*) for *Dtab.i.x.j* in *B.i.j* (changes (e) and (f)), when the predicate $H(j,i,r,x)$ holds for a position r in *Q.j.i*. In action *C.i.j* we update the distance *D.i.x* in the same way as in action *B.i.j* relying on invariant W_3 (changes (g), (h) and (i)). The resulting algorithm is given in Figure 6.

In this coarsement step no non-determinism has been added. The value of *Dtab.i.x.j* for a particular node x can be found on the link *Q.j.i*. Thus, removing the variable *Dtab* leaves the set of executions unchanged.

5.3. Summing up

We have shown that the routeing information algorithm of Tajibnapis basically consists of two different mechanisms, the *dsn*- and the *Dtab*-computations, that have been identified and abstracted away. However, the main purpose of the algorithm, the *D*- computation along the links *Q* still remains.

We have that $\mathcal{T} \sqsupseteq_I \mathcal{T}_2$ where I is the conjunction of the data invariants introduced in the two coarsement steps

$$I: U \wedge W$$

In order to verify $\mathcal{T}$ it remains for us to verify the correctness of $\mathcal{T}_2$ with respect to R.

```
T2 = ||[ var Q.i.j value list for i,j ∈ V; p.i.j ∈ int for i,j ∈ V;
         nbrs.i index set for i ∈ V

     D.i.i := 0 for i ∈ V; D.i.j := N for i,j ∈ V(i ≠ j);
     Q.i.j :=<> for i,j ∈ V(i ≠ j); p.i.j := -1 for i,j ∈ V(i ≠ j);
     nbrs.i := ∅ for i ∈ V

     do
[A.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q = < up > →
          p.j.i := p.j.i + 1; nbrs.i := nbrs.i ∪ {j}; D.i.j := 1;
          Q.i.a := Q.i.a, < j, 1 > for a ∈ nbrs.i, a ≠ j;
          Q.i.j := Q.i.j, < x, D.i.x > for x ∈ V, x ≠ i ∧ x ≠ j
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[B.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q = < down > →
          p.j.i := p.j.i + 1; nbrs.i := nbrs.i - {j};
          (if nbrs.i ≠ ∅
             ∧(∀k ∈ nbrs.i : ∃r ∈ int : D.i.x ≠ (snd(Q.k.i.r) + 1|H(k, i, r, x))) →
               olddist := D.i.x;
               (∃r ∈ int : D.i.x := min_{a∈nbrs.i}{N, snd(Q.a.i.r) + 1|H(a, i, r, x)});
               if olddist ≠ D.i.x → Q.i.a := Q.i.a, < x, D.i.x > for a ∈ nbrs.i fi
           elseif nbrs.i = ∅ → D.i.x := N
           fi
          ) for x ∈ V, x ≠ i
     for i,j ∈ V(i ≠ j), q = p.j.i + 1

[C.i.j] | Q.j.i.q ≠ <> ∧ Q.j.i.q ≠ < up > ∧ Q.j.i.q ≠ < down > →
          p.j.i := p.j.i + 1; < x, l > := Q.j.i.(p.j.i);
          if x ≠ i ∧ j ∈ nbrs.i →
            if D.i.x > l + 1
               ∨(∀k ∈ nbrs.i : ∃r ∈ int : D.i.x ≠ (snd(Q.k.i.r) + 1|H(k, i, r, x))) →
                 olddist := D.i.x;
                 (∃r ∈ int : D.i.x := min_{a∈nbrs.i}{N, snd(Q.a.i.r) + 1|H(a, i, r, x)});
                 if D.i.x ≠ olddist → Q.i.a := Q.i.a, < x, D.i.x > for a ∈ nbrs.i fi
            fi
          fi
     for i,j ∈ V(i ≠ j), q = p.j.i + 1
     od
  ]|: D.i.j ∈ int for i,j ∈ V
```

Figure 6. The algorithm after abstracting away information about neighbour distances

The algorithm $\mathcal{T}_2$ establishes the invariants below. First of all, the computed D-values are always between 0 and the maximum value N. Moreover, every data message contains a distance value, l, that is in the same interval.

$$J_0\text{: } (\forall i, j, x \in V, l \in int\text{: } i \neq j, x \neq i, x \neq j\text{:} \\ (0 \leqslant D.i.x \leqslant N) \wedge (<x,l> \in Q.i.j \Rightarrow 0 < l \leqslant N))$$

Secondly, a node either has the correct distance information when compared to its neighbours, or then some neighbour has sent new information to this node and this information is still in the queue of unprocessed messages:

$$J_1\colon (\forall i, j, x \in V, k \in int\colon x \neq i, x \neq j, j \in nbrs.i\colon \\ (D.i.x \leq \min\{N, D.j.x + 1\}) \vee (<x, D.j.x> = Q.j.i.k \wedge k > p.j.i))$$

Besides, a node always sends a computed distance to its neighbours:

$$J_2\colon (\forall i, j, x \in V, l \in int\colon (i, j) \in E, x \neq i, x \neq j\colon \\ (D.j.x = l \wedge l < N) \Rightarrow (\exists k \in int\colon Q.j.i.k = <x,l>))$$

The messages on the links are either *up-*, *down-* or data messages:

$$J_3\colon (\forall i, j, x \in V, k \in int\colon j \in nbrs.i, 0 \leq k \leq p.i.j\colon \\ Q.j.i.k = <up> \vee Q.j.i.k = <down> \vee \\ (\exists x \in V, l \in int\colon x \neq i, x \neq j\colon Q.j.i.k = <x,l>))$$

Furthermore, when there are no unprocessed messages left on the links, the algorithm has computed the minimum distances:

$$J_4\colon (\forall i, j \in V\colon i \neq j\colon j \in nbrs.i \wedge Q.j.i.(p.j.i + 1) = <>) \\ \Rightarrow (\forall i, x \in V\colon D.i.x = \delta.i.x)$$

Now, using the standard invariant based methods, the correctness of $\mathcal{T}_2$ with respect to the conjunction of the invariants J,

$$J\colon J_0 \wedge J_1 \wedge J_2 \wedge J_3 \wedge J_4$$

can be shown thereby establishing the correctness of the algorithm of Tajibnapis with respect to R given previously. This holds because $J \wedge \neg g\mathcal{T}_2 \Rightarrow R$, where $g\mathcal{T}_2$ denotes the disjunction of the guards in $\mathcal{T}_2$.

Comparing the algorithms

Let us compare $\mathcal{T}$ and $\mathcal{T}_2$. In $\mathcal{T}_2$ we have less variables than in the original algorithm $\mathcal{T}$. Also the code is shorter in terms of assignment statements. The number of actions is, however, the same. When inspecting the code of $\mathcal{T}_2$ carefully, we notice that the boolean conditions are more complicated than those in $\mathcal{T}$.

Considering the proofs we can state that the length of our proof using the superposition rule within the reverse engineering method is basically the same as if the standard invariant based methods were used. The advantage is, however, that our proof is carried out in small manageable steps. Also the invariants are developed stepwise.

Further simplifications

In a sense $\mathcal{T}_2$ is the simplest algorithm we can derive. There are, however, two local variables left, *nbrs* and Q. Abstracting away the *nbrs.i*-variable containing the neighbours of node i by replacing $nbrs.i = j$ with $(i,j) \in E$ is a trivial coarsement step.

The variable Q can be considered to be global and visible to some environment which sends the *up-* and the *down*-messages. However, if we ignore the environment that

generates these link faults and repairs, we get further simplifications. We could, thus, abstract away both the *up*- and the *down*-messages, and let the network be static. However, by performing these steps the computation would be changed and more general data refinement than the superposition refinement would be involved.

As a next step the links between the nodes could be abstracted away. The distances would then be directly compared to each other between neighbours. This coarsement step was used as an example of coarsement in Sections 2 and 3 of this paper. The algorithm $\mathcal{S}'$ was shown in Figure 3. Now comparing $\mathcal{T}$ and $\mathcal{S}'$ we can see a huge difference in size and in complexity.

The algorithm $\mathcal{S}'$ is the action system of the highest abstract level we can derive. However, if we are not concerned with the syntactic format of the algorithm, we can still derive the specification $\mathcal{S}''$ from $\mathcal{S}'$ as shown in Section 2. Thus, a three line specification can be derived from the algorithm of Tajibnapis. Observe that $\mathcal{S}''$ is not an action system and therefore does not have an interpretation as a parallel or distributed program in our sense.

6. CONCLUSION

We have introduced a formal approach to reverse engineering distributed systems. The result of this work is that a non-trivial distributed algorithm can be turned into a higher level specification by intermediate coarsement steps that preserve the basic behaviour of the implementation. Since each coarsement step can be formally verified within the refinement calculus, the correctness of the specification with respect to the original algorithm can be established. The refinement calculus/action systems framework provides us with a structured way of working with the algorithms and their proofs.

An additional benefit of our method is that the target algorithm becomes more structured, as it can be considered to consist of a number of layers interacting with each other. Hence, it helps in understanding and presenting a difficult algorithm. The practicality of our method still remains to be shown, but tools, which will ease its use, are being created for analysing and developing distributed systems within the action systems and the refinement calculus framework (Laangbacka, Ruksenas and von Wright, 1995).

We have analysed the algorithm of Tajibnapis using our reverse engineering method. We have given an informal correctness proof here, but all steps can be made formal as shown in (Sere and Waldén, 1994). This algorithm has been previously verified in the literature using assertional reasoning techniques (Lamport, 1982; Schoone, 1991). These approaches do not, however, give any general methodology for structuring the algorithms as ours does.

Our coarsement step is closely related to the reduction method of Lipton (1975), where a parallel program is analysed by collapsing pieces of the program together. This method is mostly used for reasoning about the code. In contrast to Lipton, the method presented here is based on a formal calculus, the refinement calculus, for reasoning about programs. The advantage of the refinement calculus is that we can work directly on program text. Its main purpose is to provide a basis for the stepwise refinement approach to program construction. Our work shows how this calculus can be used to create a specification for an existing algorithm.

The idea of using program transformations within reverse engineering is not new. It has been previously used in e.g. Ward (1993); Breuer and Lano (1991). Our work

shows, however, how program transformations and the stepwise coarsement approach are formalized within the refinement calculus. Furthermore, we show how these techniques are applied on distributed systems. Also layers have been previously used in the literature of distributed algorithms, but only as a design tool (Stomp, 1989; Elrad and Francez, 1982).

Acknowledgements

The authors would like to thank the Programming Methods Group at Aabo Akademi University for comments. The work reported here has been carried out within the Irene-project supported by the Academy of Finland.

References

Back, R. J. R. (1978) 'On the correctness of refinement in program development', Ph.D. thesis, Report A-1978-4, Department of Computer Science, University of Helsinki, Finland.

Back, R. J. R. and Kurki-Suonio, R. (1983) 'Decentralization of process nets with centralized control', *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ACM, New York, pp. 131–142.

Back, R. J. R. and Sere, K. (1993) 'Superposition refinement of reactive systems', Series A-144, Reports on Computer Science and Mathematics, Aabo Akademi University, Finland. To appear in *Formal Aspects of Computing*.

Breuer, P. T. and Lano, K. (1991) 'Creating specifications from code: reverse engineering techniques', *Journal of Software Maintenance: Research and Practice*, **3**, 145–162.

Brown, A. J. (1993) 'Specifications and reverse-engineering', *Journal of Software Maintenance: Research and Practice*, **3**, 147–153.

Chandy, K. M. and Misra, J. (1988) *Parallel Program Design: A Foundation*, Addison-Wesley Publishing Company, New York.

Chikofsky, E. J. and Cross II, J. H. (1990) 'Reverse engineering and design recovery: A taxonomy', *IEEE Software*, 13–17 January 1990.

Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice-Hall International, Englewood Cliffs, New Jersey.

Elrad, T. and Francez, N. (1982) 'Decomposition of distributed programs into communication-closed layers', *Science of Computer Programming*, **2**, 155–173.

Francez, N. and Forman, I. R. (1990) 'Superimposition for interacting processes', in Baeten, J. C. M. and Klop, J. W. (Eds), *Proceedings of CONCUR '90 Theories of Concurrency: Unification and Extension*, LNCS 458, Amsterdam, The Netherlands, August 1990, Springer-Verlag, Berlin Heidelberg, pp. 230–245.

Katz, S. M. (1993) 'A superimposition control construct for distributed systems', *ACM Transactions on Programming Languages and Systems*, **2**, 337–356.

Lamport, L. (1982) 'An assertional correctness proof of a distributed algorithm', *Science of Computer Programming*, **2**, 175–206.

Lipton, R. J. (1975) 'Reduction: A method of proving properties of parallel programs', *Communications of the ACM*, **12**, 717–721.

Laangbacka, T., Rukšėnas, R. and von Wright, J. (1995) 'TkWinHOL: A tool for window inference in HOL', in Schubert, E. T., Windley, P. J. and Alves-Foss, J. (Eds), *Proceedings of Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, LNCS 971, Aspen Grove, Utah, USA, September 1995, Springer-Verlag, Berlin Heidelberg, pp. 245–260.

Morgan, C. C. (1988) 'The specification statement', *ACM Transactions on Programming Languages and Systems*, **3**, 403–419.

Morgan, C. C. (1994) *Programming from Specifications*, 2nd edn. Prentice-Hall International, (UK) Limited, Hemel Hempstead, UK.

Morris, J. M. (1987) 'A theoretical basis for stepwise refinement and the programming calculus', *Science of Computer Programming*, **9**, 287–306.

Schoone, A. A. (1991) 'Assertional verification in distributed computing', Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands.

Sere, K. (1990) 'Stepwise derivation of parallel algorithms'. Ph.D. thesis, Department of Computer Science, Aabo Akademi University, Turku, Finland.

Sere, K. and Waldén, M. (1994) 'Verification of a distributed algorithm due to Chu', Series A-156, Reports on Computer Science and Mathematics, Aabo Akademi University, Finland. Abstract appeared in *Proceedings of The 13th Annual Symposium on the Principles of Distributed Computing*, Los Angeles, California, USA, ACM, New York, p. 391.

Stomp, F. A. (1989) 'Design and verification of distributed network algorithms: foundations and applications', Ph.D. thesis, Department of Computer Science, Eindhoven University of Technology, The Netherlands.

Tajibnapis, W. D. (1977) 'A correctness proof of a topology information maintenance protocol for a distributed computer network', *Communications of the ACM*, **7**, 477–485.

Udink, R. (1995) 'Program refinement in UNITY-like environments', Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands.

Ward, M. (1993) 'Abstracting a specification from code', *Journal of Software Maintenance: Research and Practice*, **2**, 101–122.

Authors' biographies:

Kaisa Sere, Ph.D., is an associate professor of computer science at the University of Kuopio, Department of Applied Mathematics and Computer Science.
She took her M.Sc. exam in mathematics and Ph.D. in computer science at the Aabo Akademi University. Her current research interests are programming methodology, formal methods, parallel and distributed computing, VLSI-design, and neural networks.

Marina Waldén, M.Sc., is a researcher in the Computer Science Department at Aabo Akademi University. She is currently working on her doctoral thesis.
She took her M.Sc. exam in computer science at Aabo Akademi University. Her major research interests are formal methods and distributed computing.